2009

Modélisation d'un réseau ferroviaire

 # Familiarisation avec la structure de graphe.
 # Utilisation de fonctions de dessin 2D OpenGL.
 # Codage des algorithmes fondamentaux des graphes et de synthèse d'images.
 # Découverte d'une API d'interface utilisateur : GLUI.

Etronnier-Renaud Leslie ----------
Margheriti Sébastien ----------
Issartel Julien ----------

28/05/2009

Introduction

Dans le cadre de notre second semestre en première année à l'école d'ingénieur IMAC, il nous a été demandé de réaliser un projet de grande ampleur consistant à la réalisation d'un réseau ferroviaire. Deux domaines sont réunis pour réaliser ce projet, le premier étant l'algorithmie avancée, et le deuxième la synthèse d'image.

Notre travail se divise en deux parties, une plus dirigée vers le rendu, et l'autre vers les algorithmes. Nous devons donc mettre en place un programme qui permettra d'afficher un réseau ferroviaire, sur une carte faite par nos soins, et coder des algorithmes fondamentaux pour résoudre les principales problématiques qui peuvent se poser lors de la création d'un réseau.

Pour ce faire, nous commencerons par positionner les enjeux et les problématiques du projet, expliquer nos besoins et les résultats que nous souhaitons obtenir, pour ensuite décrire la structure de notre projet, la justification du découpage des fichiers et de nos fonctions. Nous détaillerons ensuite plus en détail le projet en lui-même. Nous parlerons ainsi des choix faits et du morcellement des fichiers de notre projet, puis nous parlerons ensuite des structures implémentées ainsi que des algorithmes qui les exploitent. Ensuite, nous nous consacrerons un chapitre sur l'aspect interactif et graphique du projet, durant lequel nous expliquerons son fonctionnement, ainsi que la façon dont nous l'avons réalisée. Nous finirons par exposer les résultats de notre projet, à savoir à quoi il ressemble, et avec quelle efficacité tout ce que nous avons implémenté fonctionne.

Sommaire

I. Positionnement du problème 4
A. Quels enjeux ? 4
B. Les difficultés à gérer 4
II. Description et structuration du projet 5
A. Description globale du fonctionnement de l'application 5
B. Conception et structuration des fichiers 6
III. Structures et Algorithmes 7
A. Structuration 7
B. Algorithmes 8
1. Parcours en profondeur et en largeur 8
2. Changements de gare 9
3. L'algorithme de Bresenham 10
4. Plus court chemin entre deux gares 11
5. Pvc 12
6. 2-opt 13
7. Auto-accessibilité 14
IV. Application interactive et Guide de l'Utilisateur 15
A. Réalisation de l'application interactive 15
1. Gestion souris-Clavier 15
2. Affichage du réseau 15
3. L'interface Glui 16
B. Guide de l'utilisateur 18
1. Pour commencer 18
2. Afficher 19
3. Edition 20
V. Résultats obtenus par notre application 21
A. Les calculs 21
1. La carte 21
2. Le plus court chemin 22
3. Le changement de Gares 23
4. Le PVC 24
5. Le calcul du kilométrage total 25
6. Gares auto accessibles 26
B. L'édition 26

I. Positionnement du problème

A. Quels enjeux ?

L'enjeu principal de notre projet est de réaliser la simulation d'un réseau ferroviaire sous forme logicielle, qui en permettra une gestion virtuelle. Pour ce faire, le logiciel devra fournir les outils de calculs et de détermination de chemin essentiels, pour répondre aux principaux problèmes d'un réseau ferroviaire. Afin d'être utilisable de façon plus générique, ce logiciel devra aussi permettre l'édition du réseau ferroviaire, à savoir l'ajout de gares, de villes, de rails ainsi que leur suppression, pour répondre aux problèmes relatifs au cas de figure de l'utilisateur.

Enfin, de façon à ce que ce logiciel soit plus ergonomique et utilisable, il devra permettre une gestion graphique et minimiser l'utilisation de la ligne de commande.

B. Les difficultés à gérer

On s'aperçoit ainsi rapidement que le projet demandera un certain nombre d'opérations sur le réseau. Sa représentation « virtuelle » devra donc être faite via une structure robuste, que nous maitrisons parfaitement.

Le programme, en disposant d'un système d'édition, proposera de plus une utilisation au cas par cas. Ainsi, tout ce qui sera implémenté dans le programme devra fonctionner à tous les coups. A l'implémentation de chaque fonctionnalité, il faudra ainsi prendre en compte tous les cas possibles et imaginables, et essayer de programmer de façon la plus « générique possible ».

Nous aurons de plus un certain nombre d'algorithmes à implémenter. Si nous avons déjà une idée de leur fonctionnement logique, les mettre sous forme de code (qui fonctionne !) semble être une toute autre tâche.

Enfin, le programme propose une gestion graphique de l'application. Il faudra ainsi gérer en permanence les relations « homme-machine », et retranscrire tout ce qui est demandé par interface (souris, clavier, affichage graphique) à la machine. Il faudra aussi retranscrire de façon agréable et compréhensible pour l'utilisateur les données stockées par l'ordinateur (représentation du graphe, des systèmes de rails, etc.).

II. Description et structuration du projet

A. Description globale du fonctionnement de l'application

Notre application est donc une application interactive qui propose plusieurs fonctionnalités à son utilisateur. L'affichage de l'application se fait grâce à la bibliothèque GLUT pour la partie purement graphique (openGL) et de la bibliothèque GLUI pour la mise en place des contrôles (boutons, zone de saisie,...). Ces derniers permettent une meilleur interactivité avec l'utilisateur que, par exemple, l'utilisation du terminal.

Au lancement du programme, seul la fenêtre de contrôle GLUI s'ouvre. L'utilisateur peut ensuite soit charger les fichiers de configurations par défaut, soit indiquer le chemin d'autres fichiers.

Les fichiers de configuration sont les fichiers:

-gares.gar qui contient les villes et les gares (nous avons choisis de ne gérer qu'une seule gare par ville. Néanmoins, une ville peut ne pas contenir de gares)

-rails.ral qui contient les liaisons entres les différentes gares

-map.ppm qui est une image ppm en niveau de gris et qui représente le terrain à afficher

Dans un premier temps, l'application va lire les fichiers. Si ils sont cohérents (vérification par exemple qu'un rail ne passe pas par la mer), elle va construire deux objets : la liste des villes et le graphe qui va représenter notre réseau. Une fois le graphe chargé, l'application va afficher le terrain. Ensuite, le programme va parcourir le graphe et dessiner la position des villes et des gares, écrire leurs noms et puis les relier entre eux, grâce à la fonction de tracer de droite d'openGL entre deux points.

Maintenant que l'ensemble de l'application est construite, le programme va ''écouter'' chaque clic de la souris sur les boutons ou sur la carte, et lancer l'algorithme ou la fonction qui correspond.

B. Conception et structuration des fichiers

Durant tout le développement du projet nous avons porté une attention particulière à la conception de celui-ci. Nous avons séparé au maximum les données, réalisé de petites fonctions pour éviter tout redondance dans le code et fortement privilégier l'utilisation des structures.

Voici la description de chacun des fichiers :

-gare.c/gare.h : contient la structure gare, qui comprend son indice, son nom, l'indice de la ville et un pointeur vers la ville dans laquelle elle se trouve, ses coordonnées (ce qui va permettre de facilement l'afficher sur la map) et ainsi qu'un flag ''visite'' qui sera utile dans plusieurs algorithmes.

En ce qui concerne les fonctions, ce sont des fonctions ''utilitaires'' qui permettent de récupérer une gare à partir de l'id d'une ville, une gare à partir de son id, d'ajouter une gare dans le graphe ou bien encore de savoir si une gare est desservie ou non.

-ville.c/ville.h : contient la structure ville, qui comprend son indice, son nom, le nombre de gares qu'elle abrite (0 ou 1) ainsi que sa position sur la map.

En ce qui concerne les fonctions, ce sont des fonctions ''utilitaires'' qui permettent de récupérer une ville par son id ou d'ajouter une ville dans le graphe.

-formes.c/formes.h : ces fichiers comportent des fonctions qui permettent de dessiner des formes primitives telles que des cercles.

-image.h/image.c : contient la structure image, qui comprend sa largeur et sa hauteur, la taille totale des pixels et le tableau de pixels.

Les fonctions, permettent d'écrire ou lire une image au format ppm(P5), de renvoyer la valeur d'un pixel grâce à ses coordonnées, de calculer la distance entre deux points (algorithme de Bresenham), de détecter la mer entre deux points ou bien encore de vérifier la cohérence des pixels de l'image.

-listeDouble.h/listeDouble.c : ces fichiers permettent de se servir de la structure de liste doublement chaînée que nous avons utilisée pour le graphe. Ils comportent les fonctions classiques de parcours de liste, placement en tête de liste, etc. Pour que cette structure de donnée soit facilement réutilisable, nous avons déclaré en void * l'objet que contient la liste. Ainsi, nous pouvons placer n'importe quel type d'objet dans notre liste, sans avoir à retoucher le code.

-utils.c/utils.h : comporte divers fonctions de traitement des chaînes de caractères (principalement utilisées lors de la lecture d'image pour passer les commentaires).

-point.h : structure permettant de représenter un point 2D.

-graphe.h/graphe.c : ces fichiers contiennent la structure de représentation de notre graphe, ainsi que la majorité de nos algorithmes. Ils seront détaillés dans la partie suivante de ce rapport.

-main.c : c'est le fichier principal de notre application. C'est ici que la partie openGL et l'interface GLUI sont gérés.

III. Structures et Algorithmes

A. Structuration

Notre application doit donc être capable de gérer des connexions entre plusieurs villes et gares, de réaliser des calculs sur les distances, de trouver le plus court chemin entre deux gares, en somme tout un ensemble de fonctionnalités qui nécessitent une structuration robuste. La structure la plus adaptée à ces problématiques est le graphe.

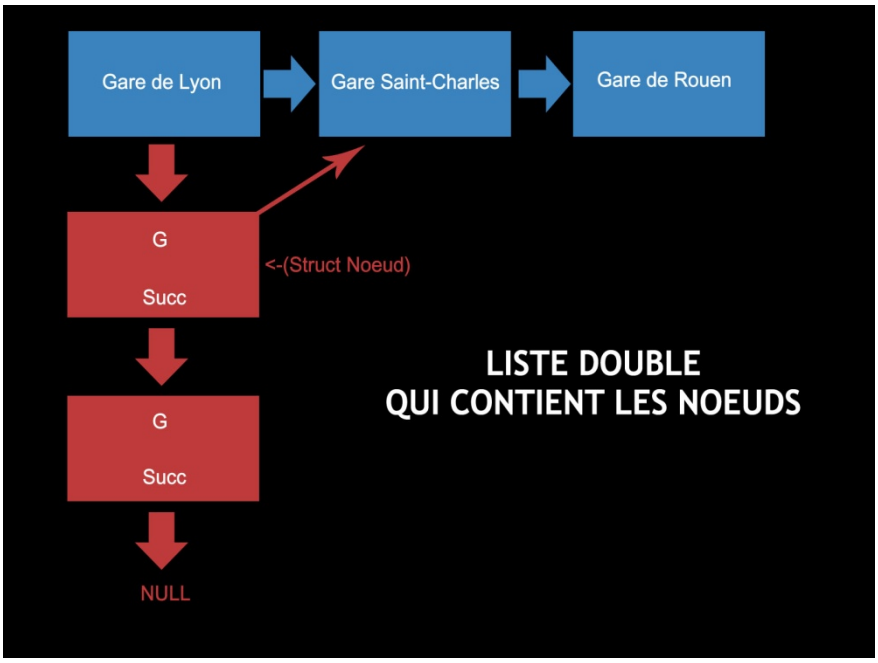
Nous avons donc du implémenter cette structure de données. L'implémentation et les algorithmes qui utilisent le graphe sont dans les fichiers ''graphe.c'' et ''graphe.h''.

Étant donné qu'il nous a été demandé de pouvoir rajouter des nœuds à la volée dans notre graphe, nous avons donc opté pour une représentation par pointeur. Même si celle-ci est un peu plus complexe à concevoir et à manipuler qu'une représentation FS-APS, elle a l'avantage d'être plus souple car elle utilise un système de listes.

Grâce à la représentation par pointeur, il nous est très facile de rajouter une gare à notre graphe : il suffit d'insérer un élément en queue de liste grâce à la fonction approprié. Si nous avions utilisé une structure FS-APS, cela aurait était beaucoup plus compliqué, étant donnée qu'il n'est pas aisé d'agrandir un tableau.

Notre graphe se compose d'une liste doublement chaînée de Nœuds. Un Nœud est une structure (définie dans graphe.h) qui contient un pointeur vers une gare, un entier pour enregistrer la distance et un pointeur vers le nœud successeur.

Le premier nœud de chaque cellule de la liste comporte l'ensemble des gares du réseau, ce sont les nœuds principaux. Et chacun de ces nœuds pointent vers le nœud successeur du nœud principal (c'est à dire un nœud au quel il est relié), qui va pointer vers le second successeur du nœud principal, ainsi de suite.



La construction du graphe au chargement du programme se déroule en deux étapes:

- Tout d'abord, grâce au fichier de gares, le programme va pour chaque gare, instancier un objet gare, l'insérer dans une structure de donnée Nœud, qui elle même sera insérée dans la liste qui va représenter notre graphe. Ainsi à cette étape, nous disposons de toutes les gares de notre réseau stockées dans notre liste.

- Ensuite, grâce au fichier de rails, on va pouvoir construire les liaisons entre les gares et ainsi finir la construction de notre graphe. Pour chaque ligne du fichier, le programme récupère les deux identifiants des gares. Il va ensuite parcourir le graphe pour récupérer des pointeurs vers ces deux gares. Pour finir il va se positionner sur le nœud qui comprend la première gare et ajouter comme successeur un nouveau nœud qui contiendra le pointeur vers la seconde gare (c'est-à-dire celle de destination).

B. Algorithmes

1. Parcours en profondeur et en largeur

Il existe différentes façons de parcourir un graphe. Les deux principales sont le parcours en profondeur et le parcours en largeur. Nous avons donc implémenté ces deux parcours, qui nous seront très utiles par la suite dans certains algorithmes.

Parcours en largeur

Ce parcours consiste, à partir d'un sommet S, à lister d'abord tous ses voisins et puis ensuite les explorer un par un. Il nécessite donc une file et aussi de pouvoir ''marquer'' les nœuds par lesquels nous sommes déjà passés, pour ne pas les compter plusieurs fois. Pour la file, nous avons utilisé notre structure de liste doublement chaîné, qui comporte des fonctions permettant de l'utiliser comme une file. Pour le marquage, nous avons ajouté un champ ''visité'' à notre structure gare, ainsi on peut changer la valeur de ce flag et savoir que l'on est déjà passé par le nœud qui contient cette gare.

Voici le déroulement de l'algorithme:

->on insère dans la file le sommet

->tant que la file n'est pas vide

-> On récupère le premier élément entré dans la file

->pour chacun de ses successeurs

->si le nœud n'a pas déjà été visité

->on le met dans la file

->et on le marque comme visité

Cet algorithme est utilisé par l'algorithme qui permet de détecter si de toutes gares, on peut atteindre toute les autres.

Parcours en profondeur

Ce parcours consiste, pour chaque sommet, à prendre le premier voisin jusqu'à ce qu'un sommet n'ait plus de voisins et revient ensuite au sommet père. Ce parcours, permet un parcours récursif du graphe. Ici aussi il est important de pouvoir marquer les nœuds par lesquels nous sommes passés pour éviter de tourner indéfiniment.

Voici le déroulement de l'algorithme :

->On marque le noeud

->tant qu'il n'est pas nul

->si il n'est pas visité, on rappel la fonction avec ce noeud

->on passe au successeur

Cet algorithme est utilisé dans plusieurs endroits du programme, par exemple pour calculer le kilométrage total du graphe (parcoursProfondeurKiloTotal) ou bien encore pour compter le nombre de rails (parcoursProfondeurNbRails).

2. Changements de gare

L'algorithme de changements de gare consiste à trouver, à partir d'une ville donnée, quelles villes sont accessibles en n changements. Ainsi, l'utilisateur doit choisir une ville, un certain nombre, et verra apparaître sur la carte les villes où il peut se rendre.

L'algorithme

Cet algorithme utilise le parcours en largeur qui a été décrit ci-dessus. Il a donc fallut reprendre cet algorithme et changer quelques lignes pour l'adapter à notre cas.

Tout d'abord, il nous faut deux piles. Elles permettront de garder les nœuds que l'on a visités. La première pile sert à empiler les nœuds que nous visitons, tandis que la deuxième contient les successeurs de ces derniers.

Au tout début, la fonction empile le premier nœud dans la pile F. Puis, on empile dans la deuxième pile Save, tous les successeurs des nœuds de F. Une fois la pile F vidée, on copie la pile Save dans F, et l'on recommence ainsi jusqu'à ce que la pile F ne contienne plus rien. Cette méthode implique une double boucle *while,* les deux ayant pour condition que la pile F ne soit pas vide.

Ceci diffère donc assez légèrement du parcours en largeur, mais garde un principe similaire.

Vient ensuite la question des changements de gares. Un compteur est initialisé au début de la fonction à -1. A chaque passage dans la première boucle *while*, le compteur est incrémenté. Puis, une fois rendu dans la deuxième boule *while*, on teste si le compteur est égal à nombre *n* rentré par l'utilisateur. Si tel est le cas, les nœuds sont non seulement empilés dans la pile Save, mais également inséré dans une liste Result. C'est cette liste qui sera retournée en résultat, et qui contiendra toutes les gares recherchées.

L'affichage

Pour ce qui est de l'affichage du résultat, nous avons repris les fonctions qui affichaient les villes et les gares. Lors de l'affichage des villes, si la liste contenant le résultat aux changements de gares contient quelque chose, alors chaque ville affichée et contenue dans cette liste apparaît d'une couleur différente.

3. L'algorithme de Bresenham

L'algorithme

L'algorithme de Bresenham permet de déterminer quels sont les pixels qui doivent être allumés afin de tracer une droite entre deux points. Techniquement, il consiste à calculer en tout points du segment qui relie les deux points le pixel le plus optimal à allumer pour représenter ce point, afin au final, de réaliser la droite la plus précise possible.

Implémentation

Dans notre cas, nous avons utilisé cet algorithme pour déterminer par quel pixel passe un rail et ainsi récupérer la valeur de ces pixels. Le fait de connaître la valeur des pixels par lesquels la droite passe nous a permis deux choses :

- calculer le kilométrage entre deux gares. A chaque valeur d'un pixel correspond une certaine distance. Il suffit donc de sommer les distances correspondantes de chaque pixel pour obtenir le kilométrage total du rail.
- détecter si un rail passe par la mer. Nous lançons l'algorithme et si l'on passe par un pixel qui correspond à la mer, nous retournons une valeur d'erreur.

A chaque fois, le principe de l'algorithme est le même :

Il faut tout d'abord déterminer l'orientation de la droite qui relie le point de départ à celui de l'arrivée. Suivant cette orientation, on découpe l'algorithme en deux parties :

Une pour l'incrémentation selon l'axe des x, et une pour l'axe des y. Selon l'axe de la pente, on incrémentera x, ou y, et l'on allumera le nouveau pixel, avec sa nouvelle coordonnée. Cette incrémentation se fera lorsque les valeurs réelles du point auront dépassé une certaine limite d'erreur. Lorsque cette erreur sera trop grosse, alors le pixel en question (qui contient une valeur entière) sera incrémenté en l'axe correspondant.

4. Plus court chemin entre deux gares

Notre programme doit pouvoir afficher le plus court chemin reliant deux gares. Pour ce faire, l'utilisateur sélectionne deux gares, et les rails à emprunter apparaissent alors clairement.

Dijkstra

Pour mettre en place cette fonctionnalité, nous avons utilisé l'algorithme de Dijkstra. Cet algorithme permet de calculer les plus courts chemins entre une gare et toutes les autres. Il en résulte donc un tableau de distances.

Voici les étapes à suivre pour cet algorithme. Tout d'abord, le plus important est donc le tableau de distances, il contient dans chaque cas, la distance entre la gare de départ, et gare concernée par cette case, il est initialisé à -1 partout, sauf pour la gare de départ qui est mise à 0. En plus de ce tableau, nous avons besoin de deux piles, qui permettront de stocker les gares que nous visitons. La première pile V contient les sommets que nous avons visités. Dans l'autre pile S, se trouve au début tous sommets.

Là commence réellement le travail. On commence par prendre le sommet de S qui a la distance la plus faible dans le tableau des distance (ce sera donc au premier tour le sommet de départ), on le dépile de S et on l'empile dans V. Ensuite, on parcourt tous les successeurs de ce sommet, et on teste. Si la distance vers ce successeur est plus grande que la distance vers le sommet ajoutée à la distance entre ce sommet et ce successeur, alors on remplace dans le tableau des distances sa valeur par cette dernière opération. Dans ce cas, un chemin plus court est donc trouvé, si non, rien n'est changé dans le tableau.

En plus d'obtenir toutes ces distances, nous souhaitons mémoriser le chemin qu'il faut parcourir. Pour ce faire, il suffit d'ajouter un tableau de prédécesseurs. Il contiendra, pour chaque gare, l'indice de la gare précédente la plus proche. En pratique, dès que l'on trouve un chemin plus court, que l'on met à jour le tableau des distances, alors, dans le tableau des successeurs, on ajoute l'indice du sommet visité dans la case concernant le successeur traité. De cette façon, pour retrouver le chemin, il suffit de prendre la ville d'arrivée choisie, et de remonter le chemin grâce aux indices.

L'affichage

Ce chemin doit être bien affiché sur la carte, on doit donc voir les rails concernés apparaître de couleur différente. Nous avons rajouté dans la fonction qui dessine les rails, une condition. Si la liste contenant les gares du plus court chemin n'est pas vide (donc que la fonction a été appelée), alors les rails reliant les gares concernées sont affichés d'une couleur différente.

5. Pvc

Principe et fonctionnement

A quoi sert-il ?

Le Problème du Voyageur de Commerce (PVC) tient son explication dans son nom. A l'époque où les voyages et transports étaient coûteux en temps (et donc en argent), les voyageurs de commercent qui travaillaient sur plusieurs villes cherchaient à trouver le meilleur itinéraire qui leur permettait de faire leur parcours en un temps optimal. Ce problème, plutôt simple lorsque peu de villes étaient à rejoindre, se complexifie exponentiellement pour devenir tout simplement insoluble à partir d'un certain nombre de villes. Même aujourd'hui, il est impossible pour un ordinateur, en un temps raisonnable, de calculer exactement le parcours optimal pour 20 villes (on mettrait 1928 ans !). C'est parce qu'on ne peut trouver (pour le moment) une solution exacte que l'on s'est rabattu sur des algorithmes d'approximation, comme celui du PVC.

Comment fonctionne t-il ?

Le principe de fonctionnement du Problème du voyageur de commerce est simple. Il consiste, dans un graphe, à partir d'un des sommets que l'on désire relier, puis de comparer les distances qui le sépare des ses successeur, pour prendre la plus courte. On se positionne ensuite sur le nouveau successeur, et l'on réapplique à ce successeur la même chose. Et ceci en faisant attention de ne pas passer deux fois par le même chemin, jusqu'à que tous les nœuds que l'on désirait relier soient parcourus.

Implémentation

La première chose, comme tout algorithme, est de vérifier si les éléments passés en paramètre sont existants. S'il n'y a qu'une gare sélectionnée, alors l'algorithme n'a pas lieu d'être. S'il y en a deux, Dijkstra semble plus indiqué et est lancé.

A partir de trois villes sélectionnées, l'algorithme se lancer. Il commence par se positionner sur la première gare prise en paramètre.

On créée ensuite un tableau qui contiendra la suite d'identifiants de gares symbolisant le chemin. On met l'identifiant du premier nœud sur la première case du tableau. On initialise tous les nœuds du graphe en tant que « non visités ».

Puis, on créée une boucle qui va visiter tous les nœuds dont les identifiants sont compris dans le tableau de visites, et on vérifie s'ils sont on déjà été visités ou pas (grâce à leur variable de visite).

S'ils ont été tous visités, on renvoie le tableau contenant le chemin. Dans cette boucle, on procède à l'algorithme en tant que tel :

On parcourt les fils de chaque nœud, et on compare les distances contenues par chaque arc entre elles (qui contient la valeur de la distance). On retient ensuite l'arc qui contient la distance la plus courte, puis on se positionne sur le nœud pointé par cet arc. On sort de la boucle, puis on y rerentre pour ce nouveau nœud.L'algorithme se déroule ainsi jusqu'à ce que tous les nœuds de la première boucle soient parcourus.

6. 2-opt

Principe et fonctionnement

A quoi sert-il ?

L'algorithme dit 2-opt est un algorithme d'optimisation. Dans notre cas, il récupère le résultat du PVC, puis essaie d'en optimiser le résultat.

Comment fonctionne-t-il ?

Comme dit précédemment, il récupère le résultat d'un algorithme, en permute successivement les membres pour déterminer si un meilleur résultat n'est pas possible. Une explication de son fonctionnement, adapté à notre cas, est donnée juste après.

Implémentation

Utiliser le 2-opt dans sa forme la plus pure et la plus optimisée est un gros challenge.

En effet, il fonctionne de cette façon :

> *Je prends le tableau créé par 2-opt, et je procède à mon algorithme.*
> *Dans une double boucle, je me balade deux fois dans mon tableau de cette façon :*
> *For (i=0;i<taille;i++)*
> *for (j=i+1;j<taille;j++)*
>
> *J'échange dans cette boucle les trajets, ce qui veut dire que je n'échange pas les directement.*
> *Soient des villes qui selon un parcours sont comme ça :*
> *0->1->2->3->4->5->6->7->8->9*
>
> *Alors si j'échange entre 2 et 7 je fais ce type d'échange :*
> *0->1->7->6->5->4->3->2->8->9*
>
> *Algorithmiquement parlant :*
> *Soient p1 à inverser avec p2*
> *RENVERSEMENT = élément_avant_p1 point sur p2, p1 point sur element_apres_p2*
> *gain = dist(p1,element_après_p1)+dist(p2, element_apres_p2)-dist(élément_avant_p1, p2)-dist(p1, point sur element_apres_p2)*
>
> *Donc l'algo serait en gros :*
> *For (i=0;i<taille;i++)*
> *for (j=i+1;j<taille;j++)*
> *RENVERSEMENT*
> *si le gain est <0, alors le chemin est plus court et il faut garder le renversement*

L'algorithme au jour ou nous écrivons n'est pas totalement implémenté. Cependant, nous avons dans un souci de simplicité enlevé deux optimisations :

- Celle du renversement qui consiste à une permutation pure et simple des deux gares
- Celle du gain, nous calculons à chaque étape la distance totale du parcours, et non le gain local réalisé par la permutation.

Nous somme conscients de cette entorse, mais nous avons estimé qu'il faut mieux proposer un algorithme qui fonctionne, même s'il n'est pas optimisé, que rien du tout.

7. Auto-accessibilité

Principe et fonctionnement

A quoi sert-il ?

Cet algorithme semble être un algorithme essentiel dans la mise en place d'un réseau. Il est en effet impensable de créer un réseau qui ne permet pas à chaque gare d'être reliée à toutes les autres.

Ainsi, pour l'utilisation de certaines fonctions, il est utile de savoir si de n'importe quel gare où on lance la fonction, il est possible d'atteindre toutes les autres gares. Une gare est dite auto-accessible s'il est possible d'atteindre toutes les autres gares à partir d'elle-même. Un réseau est auto accessible, si de toutes les gares du réseau, il est possible d'atteindre toutes les autres gares du réseau.

C'est l'algorithme que nous avons voulu mettre en place.

Comment fonctionne t-il ?

Le principe de l'algorithme n'est pas très complexe. Il suffit de parcourir chacun des éléments du graphe, de faire un parcours, et de vérifier à la fin de chaque parcours si l'on a bien traversé tous les nœuds du graphe. Si tel est le cas, alors le graphe est dit auto-accessible. Sinon, cela veut dire qu'au moins un des nœuds ne peut pas atteindre tous les autres. Et donc que le graphe n'est pas auto-accessible.

Implémentation

L'implémentation n'a pas été très complexe. Nous avons réalisé deux fonctions, une de parcours en largeur, et une comprenant l'algorithme à proprement parler.

La fonction de parcours en largeur est quasiment la même qu'un parcours en largeur normal, sauf qu'elle a été dotée d'un compteur qui s'incrémente à chaque traversée de ville. Elle renvoie la valeur du compteur.

La fonction d'auto-accessibilité en elle-même parcourt chaque nœud du graphe. Pour chaque nœud, elle lance le parcours en profondeur, et récupère la valeur renvoyée. Elle compare cette valeur renvoyée avec la taille du graphe. Si la valeur est inférieure à la taille du graphe, cela veut dire que le nœud en question n'est pas auto-accessible (il ne peut pas accéder à tous les autres nœuds), et donc que le graphe n'est pas auto accessible. La fonction renvoie immédiatement 1 (qui correspond à la valeur : « non accessible »). A la fin de l'algorithme, si tous les nœuds ont été visités, et si chacun d'eux est auto-accessible, alors la fonction renvoie 0.

IV. Application interactive et Guide de l'Utilisateur

A. Réalisation de l'application interactive

1. Gestion souris-Clavier

Fonctionnement

L'interaction entre l'utilisateur et le programme passe entièrement par la souris. L'utilisateur aura soit à cliquer sur des boutons, soit directement sur la carte elle-même. A aucun moment il n'aura besoin de passer par la console pour entrer une commande. Ceci permet de simplifier grandement l'utilisation de notre application et la rend disponible à un plus large public.
Le clavier est quand à lui pratiquement inutilisé. Seule la touche échap ou q permettent de quitter l'application.

Implémentation

Il existe deux types événements souris. Celui généré lorsqu'on clic sur un bouton GLUI, et celui généré lorsqu'on clic sur la carte.
Les clics sur la carte sont récupérés par la fonction glut void mouse(int button, int state, int x, int y), qui intercepte quel bouton a été cliqué et les coordonnées de la souris au moment du clic. Ainsi en récupérant les coordonnées du clic, nous pouvons récupérer dans le tableau de pixel de l'image, la valeur du pixel qui a été cliqué. Une fois la valeur du pixel récupérée, nous pouvons déterminer s'il s'agit d'un terrain vierge ou bien si la valeur du pixel est supérieure à 150 d'une ville. S'il s'agit d'une ville, on peut trouver simplement son indice en faisant une soustraction par 150.
Une fois l'indice récupérer, grâce aux fonctions dont disposent les fichiers gares.c et ville.c nous pouvons retrouver dans le graphe cette ville et éventuellement la gare qu'elle contient. Par la suite, l'action demandée par l'utilisateur sera appelé.

2. Affichage du réseau

L'affichage de la carte et du réseau se fait grâce aux fonctions de dessins d'openGL.
Au niveau de la carte, le programme dessine pour chaque pixel de l'image de configuration un carré dont la couleur dépend de la valeur du pixel. Les dimensions de ce carré dessiné sont par défaut de 2*2 mais il est possible de changer facilement ces valeurs qui sont déclarées en variables globales de notre programme. OpenGL nous permet simplement de dessiner un carré grâce à GL_QUADS et aux coordonnées des 4 coins du carré.

En ce qui concerne les gares et les villes, ils sont représentés respectivement par un cercle blanc et un carré noir. Le cercle est tracé grâce à une fonction que nous avons précédemment développée en TP de synthèse d'image.
Lorsque le graphe a été construit, les coordonnées des villes et des gares ont été enregistré. Ainsi, pour afficher les gares et les villes, il suffit de parcourir le graphe, récupérer la position et dessiner à cet endroit là. Nous avons aussi décidé d'afficher le nom des gares et des villes. Le principe est donc le même, on récupère le nom de la ville ou de la gare, et on l'affiche à la position de celle ci.
L'affichage des rails est quant à lui assez similaire. Le programme effectue un parcours du graphe. Il place en point de départ la position de la gare qui se trouve dans le nœud principal et pour chacun de ces successeurs il récupère la position de la gare. Ainsi, il ne reste plus qu'à tracer une simple droite

entre ces deux points. Une droite se trace facilement grâce à GL_LINES et aux coordonnées des deux points.

Nous avons aussi trouvé utile d'afficher sur la map la distance de chacun des rails. Nous l'avons inscrite au milieu de chaque rail. Pour cela il a donc fallu calculer les coordonnées du milieu du segment que représente le rail grâce à la formule géométrique suivante :

Xmilieu = (X1 +X2)/2
Ymilieu = (Y1 +Y2)/2

De plus, pour permettre de distinguer dans quelle direction se dirige un rail, nous avons rajouté au bout de ce dernier, un petit cercle gris.

3. L'interface Glui

Fonctionnement

A propos de GLUI :

Voici la définition de Wikipédia :

OpenGL User Interface Library (GLUI) est une bibliothèque en C++ qui se combine avec celle d' OpenGL utility toolkit (GLUT) et qui fournit diverses routines pour créer l'interface d'un programme entièrement avec OpenGL. L'interface est alors indépendante du système d'exploitation et du gestionnaire de fenêtres.
Cette bibliothèque, sous licence GNU LGPL, permet de compléter la bibliothèque de gestion de fenêtres OpenGL, GLUT, en apportant le support de divers éléments de contrôle tels que les boutons, les cases à cocher, les zones de texte éditables et statiques, les listes déroulantes, etc.

Glui est ainsi une interface gratuite, qui permet la création simple d'une interface qui pourra communiquer avec le programme via des boutons, champs de texte, etc.

Comment GLUI fonctionne ?
GLUI fonctionne selon un principe simple. A chaque élément d'interaction proposé par glui, est assignée une fonction qui va récupérer le signal envoyé par l'élément.
Ainsi, lorsqu'on clic sur un bouton, celui-ci va envoyer à une fonction déterminée un signal, et à ce signal, il suffit de lui faire correspondre l'action désirée.
Exemple, pour afficher « coucou » dans une console, il suffit d'assigner le bouton glui à une fonction dite de « callback », qui va récupérer le message du bouton. Cette fonction de callback va déterminer le bouton cliqué, et lancer la commande d'affichage de « coucou » dans la console.

Implémentation

Implémenter Glui n'est pas très complexe. Mais cela nécessite tout de même certaines étapes qui peuvent prendre du temps (surtout que GLUI est très peu documentée, on fonctionne beaucoup par tâtonnements).

Il faut tout d'abord créer l'élement glui, et l'affecter à la fenêtre glut correspondante :

```
GLUI *glui = GLUI_Master.create_glui( "GLUI" );
 glui->set_main_gfx_window( main_window );
```

Une fois ceci fait, suffit d'ajouter à cet élément glui ce que l'on veut utiliser. Ici, nous avons utilisé différents éléments :

GLUI_StaticText(glui,"Console de message");
Il s'agit d'un élément de texte simple. Utilisé pour présenter généralement un autre outil, il consiste à afficher un libellé sur l'interface glui.

GLUI_TextBox(glui,true,1,textbox_cb);
Il s'agit encore d'un élément de texte, mais qui prend comme source de texte un élément défini de façon externe. Il est très utile de s'en servir comme console de message par exemple. Dans ce programme, nous avons implémenté un élément de texte global, que l'on modifie à chaque fois que c'est désiré. Ainsi, dans l'interface Glui, l'affichage de cet élément se modifie à chaque modification. Nous nous en sommes servi pour remplacer tout ce qui s'affiche habituellement dans la console.

GLUI_Button(glui,"Charger une region et ses villes",1,menu_glui);
Il s'agit de l'élément «principal » de l'interface. Décrivons son fonctionnement et les paramètres qu'il prend :

- Le premier paramètre correspond à l'élément glui auquel on veut le rattacher (ici la fenêtre glui principale)
- Le deuxième correspond au libellé affiché sur le bouton
- Le troisième correspond à la valeur renvoyée par le bouton
- Le quatrième correspond à la fonction appelée par le bouton

Ces deux derniers paramètres sont importants. La fonction appelée par le bouton reçoit la valeur envoyée par celui-ci. Chaque bouton doit en renvoyer une différente. La fonction appelée détermine ainsi quel bouton est cliqué (par sa valeur), et procède (grâce à un switch-case) au traitement correspondant.

GLUI_Rollout *affichage = new GLUI_Rollout(glui, "Afficher", false);//rollout
L'outil de roll out permet de réduire ou afficher certains éléments de menu, lorsque celui est trop chargé (comme dans notre programme).

GLUI_EditText(creer, "nom de la ville:", nomVille);
Cet élément est un champ texte dans lequel on peut rentrer du texte (comme le nom d'une ville que l'on créée), et qui l'insère dans une variable string (ici nomVille).

Tout de même, GLUI étant réalisé avec du C++, nous avons du convertir à chaque fois que nécessaire les strings envoyés par glui en char, grâce à la commande (char *)nomVille.c_str(), qui caste le string en un char.

Ainsi, en plus de la création de GLUI, il a fallu créer une fonction qui réceptionne tous les messages envoyés par les boutons de GLUI, pour les traiter. La création de ce tout n'a pas été la partie la plus complexe du projet, mais elle a demandé un certain temps de réflexion, d'expérimentations et enfin d'implémentation.

B. Guide de l'utilisateur

Le guide de l'utilisateur est destiné à expliquer à un utilisateur lambda comment se servir du programme. Tout ce qui est explication du fonctionnement de ces éléments a déjà été fait dans la partie précédente.

Ici, nous vous présenterons les principales fonctionnalités du logiciel, et leur mode d'emploi. Dans une première partie, nous vous exposerons les outils de base du logiciel qui permettent de lancer une image, ou la sauvegarder. Le logiciel permet deux types d'interactions que nous qualifierons d'affichage et d'édition.

Parce qu'une capture d'écran parle mieux que des mots vains, voici à quoi ressemble l'interface «GLUI » à son lancement:

1. Pour commencer

Lors du lancement du programme, aucune carte n'est chargée. La première chose à faire est de clique sur le bouton « **Charger une région et ses villes** » après avoir rentré l'adresse des fichiers de gares, ville et carte désirés. Par défaut, les fichiers d'images se trouvent dans le dossier images/, et ceux de rails et de gares dans le dossier data/.

Une fois ceci fait, la région se charge, ainsi que ses villes et ses gares.

Juste en dessous du bouton de chargement, se trouve un bouton de **sauvegarde.** Vous pouvez à tout moment, en cliquant sur ce bouton, sauvegarder votre carte, ainsi que vos fichiers de rails et de villes. La sauvegarde remplacera les fichiers existants.

Nous tenons aussi à mentionner dans cette partie la zone de texte qui se situe tout en haut de l'application. Nous appellerons cette zone la « **console de message** ». Afin de rendre notre application indépendante de l'utilisation de la console, nous avons via cet outil affiché tous les messages que nous faisions passer habituellement dans la console. Elle affiche les instructions d'utilisation du logiciel, ou les messages d'erreur (comme « sélectionnez une seconde ville » ou « vous n'avez pas sélectionné assez de gare »).

2. Afficher

Calcul du kilométrage total

Si vous voulez savoir en un clic le kilométrage total du réseau ferroviaire, il suffit de cliquer sur ce bouton. La valeur sera affichée en kilomètres dans l'espace d'affichage juste en dessous du bouton.

Détermination des gares accessibles en « n » coups

Cet outil permet de calculer les gares accessibles en « n » coups, déterminés par l'utilisateur. Pour ce faire, il suffit de rentrer la valeur dans le champ texte contre le bouton, cliquer sur la ville de laquelle on désire partir, puis cliquer sur le bouton.

S'afficheront alors sur la carte toutes les villes qui sont accessibles au bout des « n » coups mis en paramètre.

Calcul du plus court chemin entre deux gares

Pour calculer le plus court chemin entre deux gares, il suffit de cliquer sur le bouton correspondant. Ensuite, cliquer sur la première, puis la seconde gare. Les gares traversées par ce parcours changeront alors de couleur sur la carte, vous affichant le chemin. La distance calculée quand à elle sera affichée dans la « console de message ».

Calcul du plus court chemin reliant « n » gares

Le clic droit est utilisé pour ce programme. Chaque clic droit sélectionne une ville, et la garde en mémoire. Pour sélectionner les « n gares » que vous voulez relier, faites tout d'abord un clic gauche, pour vider votre tableau de villes sélectionnées. Puis faites un clic droit sur chacune des gares que vous voulez relier. Ensuite, clique sur le bouton correspondant. Comme pour l'élément précédent, le chemin sera affiché sur la carte, par le changement de couleur de chaque gare traversée par l'application.

Déterminer et afficher du réseau accessible

Cette fonction est utile si vous désirez déterminer si le graphe est auto-accessible. Pour ce faire, c'est très simple, il suffit de cliquer sur le bouton correspondant. Dans la « console de messages » s'affichera alors la conclusion, à savoir si votre réseau est auto-accessible, ou non.

3. Edition

Ajouter une ville

Ajouter une ville est assez simple. Il suffit de mettre le nom qu'on veut donner à la ville dans la zone de texte, de cliquer sur le bouton « Ajouter une ville », puis de cliquer sur la carte où l'on veut la placer. La console de message vous expliquera si l'ajout a bien fonctionnée, ou pas. La ville s'affichera en conséquence sur la carte, en rouge, avec à côté d'elle, son nom.

Supprimer une ville

Pour supprimer une ville, il suffit de cliquer sur le bouton correspondant, puis, comme demandé dans la console de message, cliquer sur la ville que vous désirez. N'oubliez pas que pour supprimer une ville, il faut que celle-ci ne contienne pas de gare. Si tel est le cas, supprimez d'abord la gare contenue (voir ci-dessous).

Ajouter une gare

Pour ajouter une gare, il suffit de mettre le nom que vous voulez lui donner dans la zone de texte correspondant sur l'interface glui. Ensuite, cliquez sur ajouter une gare, et cliquez sur la ville à laquelle vous voulez ajouter la gare. Faites tout de même attention à ce que cette ville ne contienne pas déjà de gare. Si tel est le cas, la console de message vous le rappellera.

Supprimer une gare

Pour supprimer une gare, il faut auparavant être certain que celle-ci ne soit pas reliée à une autre gare. Si tel est le cas, supprimez d'abord le rail (voir-ci-dessous).

Sinon, cliquez simplement sur le bouton de suppression de gare, puis sur la gare de la carte que vous voulez supprimer.

Relier deux gares

Pour relier deux gares, il vous faut simplement cliquer sur le bouton correspondant, puis sur la première et la seconde gare que vous voulez relier.

Attention, les rails que l'on ajoute étant orientés, l'ordre dans lequel vous sélectionnez vos gares est déterminant. La première correspond à la gare de départ, et la seconde à la gare d'arrivée. Une petite boule blanche dessinée sur le rail symbolise l'orientation de celui-ci.

Supprimer un rail

Pour supprimer un rail, il suffit de cliquer sur le bouton correspondant à la suppression, puis de cliquer sur chacune des gares reliées.

Attention, comme dit précédemment, l'ordre de clic sur les gares est déterminant du rail que vous

V. Résultats obtenus par notre application

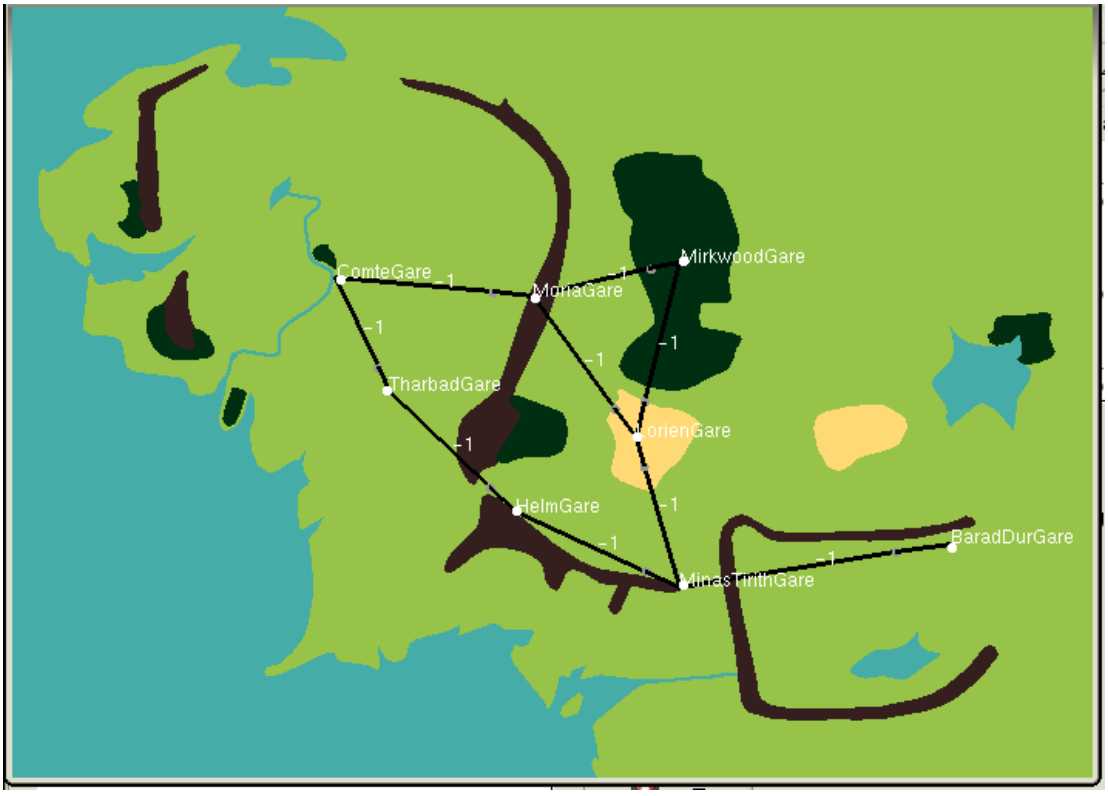
Nous allons dans cette partie vous présenter les résultats de notre travail, en présentant par des captures d'écran ce que nos algorithmes apportent, et analysant ce que nous avons réussi à faire.

A. Les calculs

1. La carte

Le réseau ferroviaire se situe dans une région, région que nous devions représenter par une carte, constituée de différents types de terrains. Nous avons souhaité une carte simple pour visualiser très bien les différents terrains et gares, et également de taille raisonnable pour ne pas ralentir les calculs et ne pas cacher l'interface GLUI.

Nous en sommes venus à obtenir deux cartes, représentant plus ou moins un même territoire. Nous avons tout d'abord une carte assez détaillée, avec des contours lisses, et d'assez grande taille. Nous affichons cette carte en vraies dimensions, ce qui permet un affichage assez agréable. Malheureusement, le détail de cette carte peut parfois amener certains ralentissements, qui peuvent parfois être très gênant. Nous avons donc repris cette carte, et en avons conçu une autre, plus petite, et moins détaillée, qui permet une utilisation optimale du programme.



2. Le plus court chemin

L'algorithme du plus court chemin permet, comme il a été dit plus haut, de trouver à partir de la sélection de deux gares, le chemin le plus court les reliant. Après avoir cliqué sur le bouton permettant de calculer ce chemin, l'utilisateur est invité à sélectionner deux gares.

Une fois le calcul demandé, un certain temps d'exécution est nécessaire pour afficher le résultat. Dans la grande carte, le temps se compte en secondes, tandis que pour la petite, l'affichage arrive en moins d'une seconde. Le temps de calcul ne varie pas suivant la longueur du chemin.

Une fois le calcul effectué, les rails formant le chemin demandé apparaissent en rouge, comme ci-dessous.

En rouge, le plus court chemin entre « ComteGare » et « MinasTirithGare »

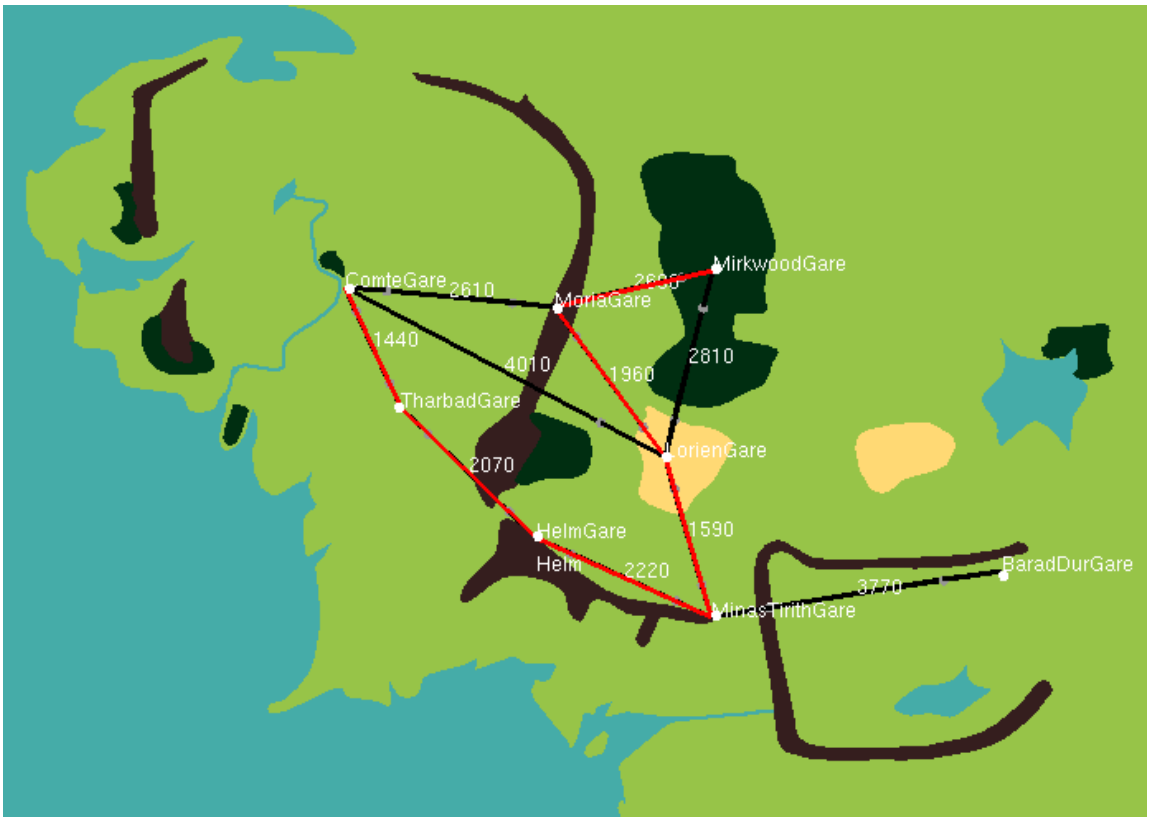
3. Le changement de Gares

Cette fonctionnalité permet d'afficher les gares accessibles après n changements depuis une gare sélectionnée par l'utilisateur. Une fois la gare sélectionnée, et le n indiqué, le temps de réponse est également assez long. Suivant la carte, ce temps est plus ou moins grand. Nous avons toujours un temps de réponse plus long pour la joli carte, mais le temps pour la plus petite n'est pas diminué de beaucoup, le temps passé à attendre est tout de même notable. A la fin du calcul, les villes résultantes sont affichées en rouge.

En rouge, les gares accessibles en 1 changement depuis ComteGare

4. Le PVC

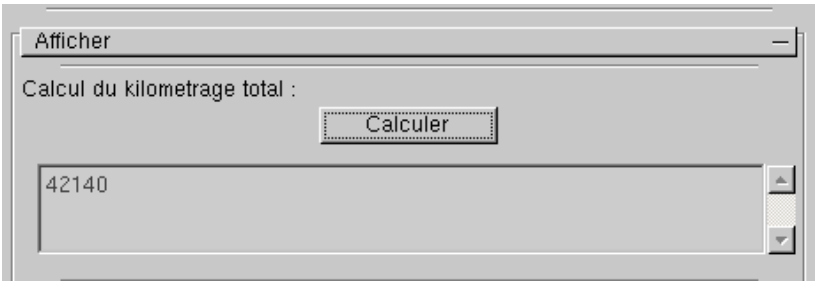
Le PVC permet de trouver le plus court trajet à effectuer passant par plusieurs gares. Dans notre programme, nous devons sélectionner avec le clic droit, au minimum trois gares, puis l'utilisateur actionne le bouton lançant le PVC, et alors le trajet obtenu s'affiche en rouge (tout comme pour le plus court chemin).

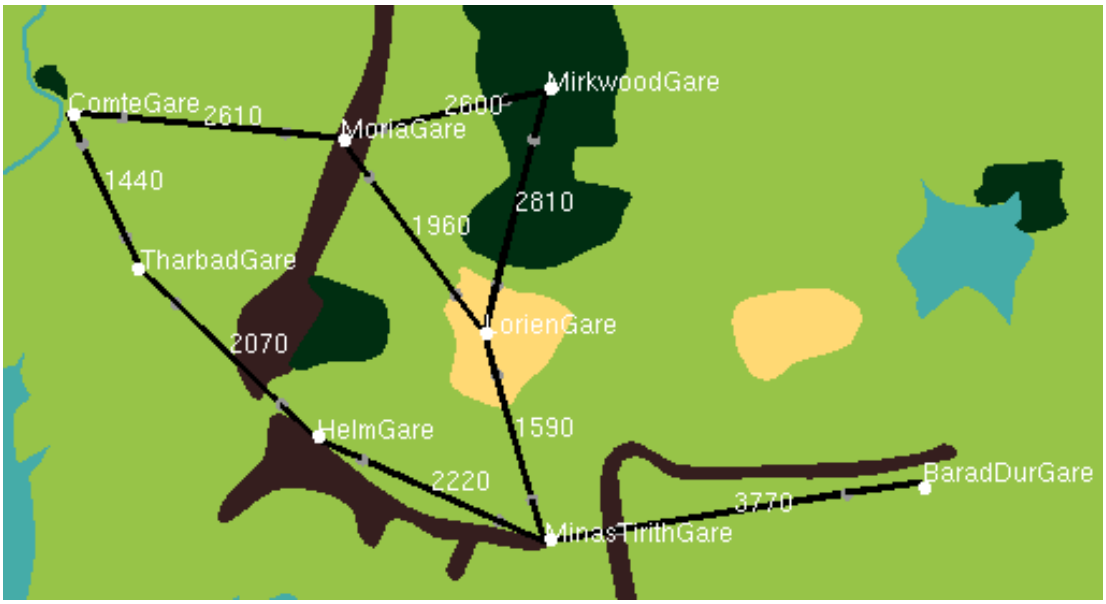


En rouge, le chemin trouvé par le pvc pour relier ComteGare, HelmGare et MirkwoodGare

5. Le calcul du kilométrage total

Le bouton GLUI « calcul du kilométrage total » permet de lancer le calcul des distances des rails. Les distances s'affichent immédiatement sur la carte, au milieu de chaque rail.





6. Gares auto accessibles

L'algorithme qui vise à vérifier si le réseau est auto accessible n'a pas de résultat visible très marquant. Il suffit d'appeler la fonction à l'aide d'une commande dans le menu, pour voir s'afficher si le réseau est auto accessible ou non.

B. L'édition

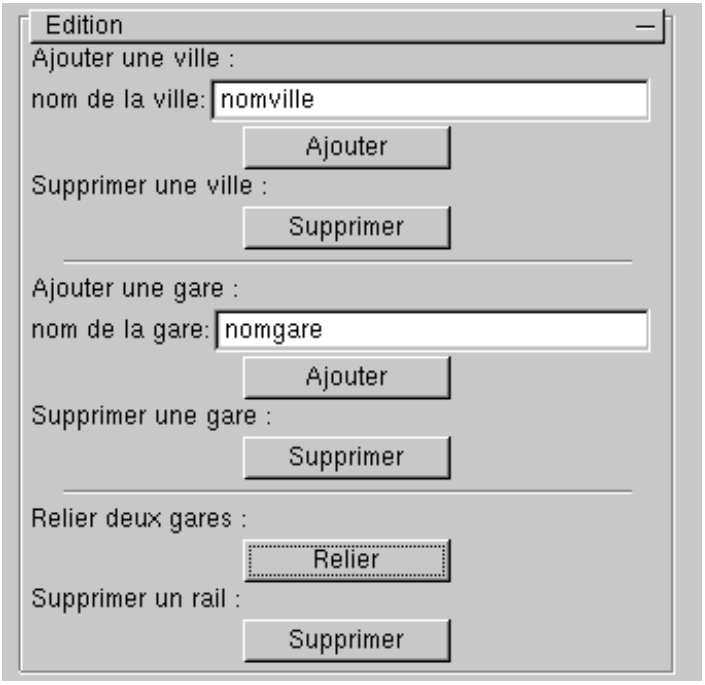
L'autre aspect de notre programme, est le côté édition. Il est en effet possible d'éditer à volonté le réseau ferroviaire. Il est possible d'ajouter des villes, des gares, des rails, ou bien de les supprimer.

L'un des aspects les plus intéressants de cette partie est la création de rails. Nous avons choisi de passer par l'interface graphique directement, et en temps réel, le rail est affiché, un bout étant positionnée sur la gare de départ, et le second sur le pointeur de la souris, suivant ses mouvements.

Ici, le rail de droite est en train d'être crée par l'utilisateur (malheureusement la souris n'a pas été conservée par le screenshot)

Le suivi de la souris par le rail à tendance à être parfois lent avec la grande map. Par contre tout est très fluide et rapide avec la map plus petite.

Parmi les autres fonctionnalités, nous avons la création d'une ville ou d'une gare, il suffit de cliquer sur l'endroit souhaité, puis d'entrer le nom dans l'interface GLUI. C'est une action qui prend effet immédiatement. La suppression des rails et des villes s'effectue en cliquant sur les villes concernées, ces actions sont également immédiates.



Conclusion

Ce projet nous a permis de mettre en pratique certaines des compétences que nous avions acquises ce semestre. Dans le domaine de l'algorithmie, nous avons pu mettre en place et coder des algorithmes fondamentaux, et les tester sur un problème réel (le réseau). Ces implémentations furent parfois très laborieuses, et nous freinèrent maintes fois dans l'avancement du projet. Et à côté de ça, nous avons mis en application nos connaissances de l'OpenGL pour pouvoir afficher de façon correcte et agréable, les résultats que nous trouvions.

Tout le projet fut effectué en trinôme, il fut donc assez difficile d'organiser nos avancées. Mais nous avons tout de même réussi à partager les tâches et le travail, pour en faire un projet équilibré selon les niveaux de chacun.

Enfin, nous pouvons dire que ce projet fut très compliqué à mettre en place, et nous n'avons malheureusement pas réussi à le terminer complètement, à intégrer toutes les fonctionnalités demandées. Certaines fonctions et algorithmes furent trop longs à mettre en place pour pouvoir correctement finir le travail, de plus, notre projet fut semé d'obstacles en tout genre qui ont allongé le temps de travail.

Il serait intéressant de pouvoir terminer totalement le projet, et le rendre encore plus captivant, en finissant tout d'abord les fonctionnalités demandées, puis en le rendant plus ludique en améliorant le graphisme de la carte, en rajoutant des possibilités, ou peut-être même un éditeur de terrain pour pouvoir soit même faire sa carte.